

Flexible Computational Science Infrastructure (FleCSI)

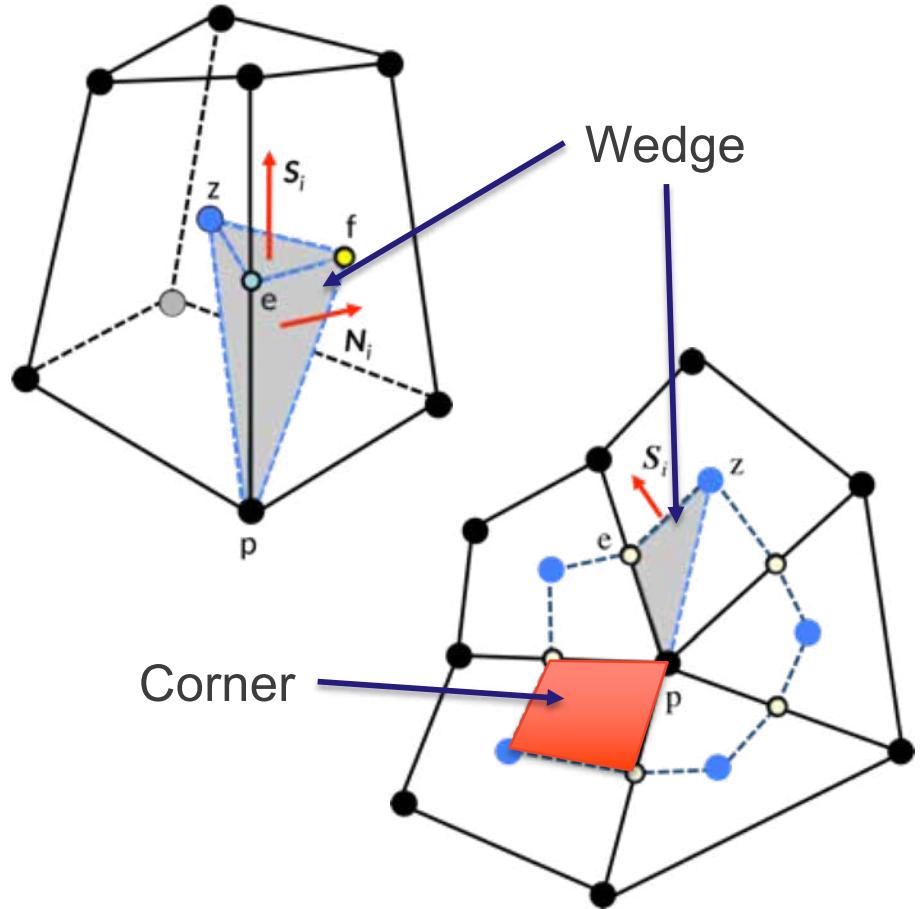
Overview & Applications Progress



Ben Bergen



Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA



- **FleCSI Overview**
 - Usability & Productivity
- **FleCSALE**
- **FleCSPH**
- **Future Work**

Contributions

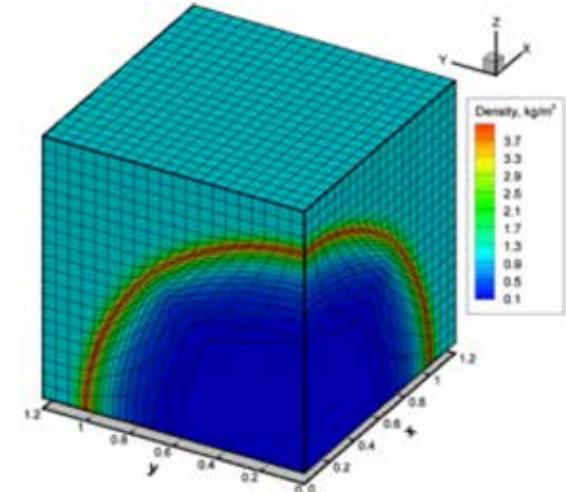
- **FleCSI Project**
 - Ben Bergen
 - Irina Demeshko
 - Nick Moss
 - Tim Kelley
 - Josh Payne
 - Navamita Ray
 - Bob Bird
 - Christoph Junghans
 - Martin Stahley
 - Ollie (Li-Ta Lo)
- **Legion Project**
 - Galen Shipman (LANL)
 - Jonathan Graham (LANL)
 - Mike Bauer (NVIDIA)
 - Sean Treichler (NVIDIA)
 - Elliott Slaughter (SLAC)
 - Alex Aiken (Stanford)
- **FleCSALE**
 - Marc Charest
 - Misha Shashkov
 - Nathaniel Morgan
 - Vincent Chiravalle
 - Mike Rogers
 - Ricardo Lebensohn
 - Pascal Grosset
 - Dave Nystrom
- **FleCSPH**
 - Julien Loiseau
 - Hyun Lim
 - Wes Even
 - Oleg Korobkin

FleCSI Overview

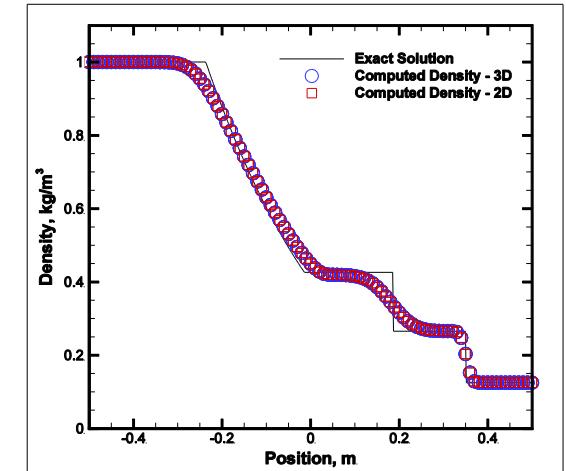
What is FleCSI?

FleCSI is a C++ programming system for developing multi-physics simulation codes...

- **Runtime abstraction layer**
 - High-level user interface, mid-level static specialization, low-level building blocks, tasking and fine-grained threading back-ends
- **Programming model**
 - Control, execution, and data models
- **Useful data structure support**
 - Mesh, N-Tree ($N=3 \rightarrow$ Octree), and KD-Tree



FleCSI: Pure 3D Lagrangian Sedov



FleCSI: 2D/3D Eulerian Sod

FleCSI Interface Structure & Programming Model

- **Low-Level (FleCSI Core Capability)**
 - FleCSI provides a templated, low-level interface that can be specialized for a particular class of physics packages and application needs
- **Specialization**
 - A specialization provides application developers with a high-level interface that is customized to their nomenclature and data structure requirements
- **Task Abstraction (FleCSI Runtime Abstraction)**
 - Using the FleCSI task abstraction layer and some compile-time techniques, the application developer is given a programming system that is transparently distributed-memory parallel
- **Kernel Abstraction (FleCSI Runtime Abstraction)**
 - Using the FleCSI kernel abstraction layer with compile-time techniques, application developers are given a programming system that is transparently fine-grained, data-parallel
- **Data Abstraction (FleCSI Core + Runtime)**
 - FleCSI provides a data model that integrates with the task and kernel abstractions to provide easy registration and access to various data types with automatic dependency tracking

FleCSI Interface Structure & Programming Model

- **Low-Level (FleCSI Core Capability)**
 - FleCSI provides a templated, low-level interface that can be specialized for a particular class of physics packages and application needs
- **Specialization**
 - A specialization provides application developers with a high-level interface that is customized to their nomenclature and data structure requirements
- **Task Abstraction (FleCSI Runtime Abstraction)**
 - Using the FleCSI task abstraction layer and some compile-time techniques, the application developer is given a programming system that is transparently distributed-memory parallel
- **Kernel Abstraction (FleCSI Runtime Abstraction)**
 - Using the FleCSI kernel abstraction layer with compile-time techniques, application developers are given a programming system that is transparently fine-grained, data-parallel
- **Data Abstraction (FleCSI Core + Runtime)**
 - FleCSI provides a data model that integrates with the task and kernel abstractions to provide easy registration and access to various data types with automatic dependency tracking

Usability & Productivity: Anatomy of a FleCSI Driver

```
flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver
```

```
flecsi_register_data_client(mesh_t, clients, m); ←  
  
flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);  
  
void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {  
    size_t count[0];  
  
    for(auto c: m.cells(owned)) {  
        p(c) = double{count++};  
    } // for  
} // initialize_pressure  
  
flecsi_register_task(initialize_pressure, loc, single);  
  
void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {  
    for(auto c: m.cells(owned)) {  
        p(c) *= 2.0;  
    } // for  
} // update_pressure  
  
flecsi_register_task(update_pressure, loc, single);  
  
void driver(int argc, char ** argv) {  
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);  
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);  
  
    flecsi_execute_task(initialize_pressure, single, mh, ph);  
  
    flecsi_execute_task(update_pressure, single, mh, ph);  
} // driver
```

Operation: Register a data client

A data client is a topology type that defines one or more index spaces, e.g., cells or nodes.

In this example:

***mesh_t* – the mesh type**

this is defined by the specialization

***clients* – the namespace**

this is an arbitrary name that is assigned by the user

***m* – the mesh instance name**

this is an arbitrary name that is assigned by the user

```

flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver

```



Operation: Register a field

A **field** is the basic type for distributed-memory array data. Fields are registered against data client types.

In this example:

mesh_t – the data client type

this is defined by the specialization

solver – the namespace

name assigned by the user

pressure – the field instance name

name assigned by the user

double – the field data type

this can be any P.O.D. or valid user-defined type

dense – the storage type

storage types allow FleCSI to choose more efficient low-level strategies for handling different types of data
(*supported types*: *dense*, *sparse*, *global*, *color*, *tuple*)

1 – the number of versions

versions allow constructs like ‘old’ and ‘new’ under a single variable name

cells – the index space

the index space upon which the variable shall be defined.

```

flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) { ←
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) { ←
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver

```

Operation: Define tasks

A FleCSI task is the basic unit of execution for distributed-memory parallelism.

In this example:

initialize_pressure and *update_pressure*

m – a mesh handle

This is a handle to a mesh data client. This type uses CRTP to inherit from the mesh type so that the mesh interface is directly available to the user.

p – a field handle

This is a handle to a field. This type exposes an access operator () that allows users to read or modify field values.

Permissions:

The permissions, *ro*, and *rw* tell the runtime what distributed-memory updates need to be performed during task execution.

```

flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) { ←
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) { ←
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver

```

Operation: Topology iterators

FleCSI automatically generates iterators for all entities defined in a topology specialization.

FleCSI also provides a mechanism for the specialization to name and expose index spaces that are associated with its entity types.

In this example:

owned – exclusive + shared

The subset of all cells that are *local* to the current color

Pre-defined subsets:

all – exclusive + shared + ghost

The subset of all cells that are defined on the current color

exclusive – I own them and nobody else cares about them

shared – I own them and other colors need them too

ghost – another color owns them and I depend on them

```

flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);           ←

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);           ←

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver

```

Operation: Register tasks

Task registration exposes all FleCSI tasks to the runtime pre-initialization, allowing them to be analyzed and processed for the low-level runtime during initialization.

In this example:

initialize_pressure, update_pressure – the task name

loc – the processor type

supported processor types are:
toc – throughput optimized core
loc – latency optimized core
mpi – MPI task

single – the launch type

supported launch types are:
single – launch as a single task
(this fits most peoples idea of task-parallelism)
index – launch as a set of tasks over an index space

```
flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver
```

Operation: Define the user driver

From the *user's point of view, this is the main function and primary control model.*

In this example:

argc – command-line argument count

argv – command-line arguments



```

flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m); ←
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver

```

Operation: Get data client handle

Get a handle to an instance of a registered data client.

In this example:

***mesh_t* – the data client type**

***clients* – the data client namespace**

must be the same as used to register the data client

***m* – the data client name**

must be a registered data client name

```
flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver
```

Operation: Get field handle

Get a handle to an instance of a registered field.

In this example:

***m* – a data client handle**

***solver* – the field namespace**

must be the same as used to register the field

***double* – the field data type**

***dense* – the storage type**

***0* – the field data version**

```

flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph); ←
    flecsi_execute_task(update_pressure, single, mh, ph); ←
} // driver

```

Operation: Execute tasks

Task execution schedules a task to be invoked by the low-level runtime.

Task-based runtimes, such as Legion, perform dynamic analysis of task dependencies to expose concurrency and optimize execution.

In this example:

initialize_pressure, update_pressure – the task name

single – the launch type

variadic arguments – task arguments

this is a task-dependent, variadic list of parameters with which to invoke the task

```
flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph); 
    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver
```

Operation: Execute tasks

The **initialize_pressure** task has read-write access to shared values in the pressure field.

```
flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver
```

Operation: Execute tasks

The *initialize_pressure* task has read-write access to shared values in the pressure field.



```
flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver
```

Operation: Execute tasks

FleCSI invokes *prologue* and *epilogue* operations around the user's task to update distributed-memory dependencies.

The prologue and epilogue functions infer which dependencies need to be updated by analyzing the permissions of the input parameters to the task.

```
flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph); ←
} // driver
```

Operation: Execute tasks

The **update_pressure** task has read access to ghost values in the pressure field.

These will be up-to-date with the results of the **initialize_pressure** task.

```
flecsi_register_data_client(mesh_t, clients, m);

flecsi_register_field(mesh_t, solver, pressure, double, dense, 1, cells);

void initialize_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    size_t count{0};

    for(auto c: m.cells(owned)) {
        p(c) = double{count++};
    } // for
} // initialize_pressure

flecsi_register_task(initialize_pressure, loc, single);

void update_pressure(mesh<ro> m, field<rw, rw, ro> p) {
    for(auto c: m.cells(owned)) {
        p(c) *= 2.0;
    } // for
} // update_pressure

flecsi_register_task(update_pressure, loc, single);

void driver(int argc, char ** argv) {
    auto mh = flecsi_get_client_handle(mesh_t, clients, m);
    auto ph = flecsi_get_handle(m, solver, pressure, double, dense, 0);

    flecsi_execute_task(initialize_pressure, single, mh, ph);

    flecsi_execute_task(update_pressure, single, mh, ph);
} // driver
```

Operation: Execute tasks

The **update_pressure** task has read access to ghost values in the pressure field.

These will be up-to-date with the results of the **initialize_pressure** task.



FleCSALE

Current Status

- **2D/3D cell-centered Eulerian and Lagrangian solvers**
 - Piecewise-constant (i.e. first-order) spatial representation
 - Second-order predictor-corrector time marching
 - Uses C++11/14 features to eliminate run-time branching based on the number of problem dimensions
 - All dimension-specific code is hidden from users
 - Multi-material support using a constant volume fraction closure
- **3D FEM Lagrangian solver**
 - Linear basis functions (i.e. second-order spatial representation)
 - Staggered grid arrangement for unstructured meshes
 - Simple strength model

Fully unstructured 2D and 3D mesh specializations developed on top of FleCSI

Mesh is templated on dimension:

2D: `burton_mesh_t<2> mesh;`

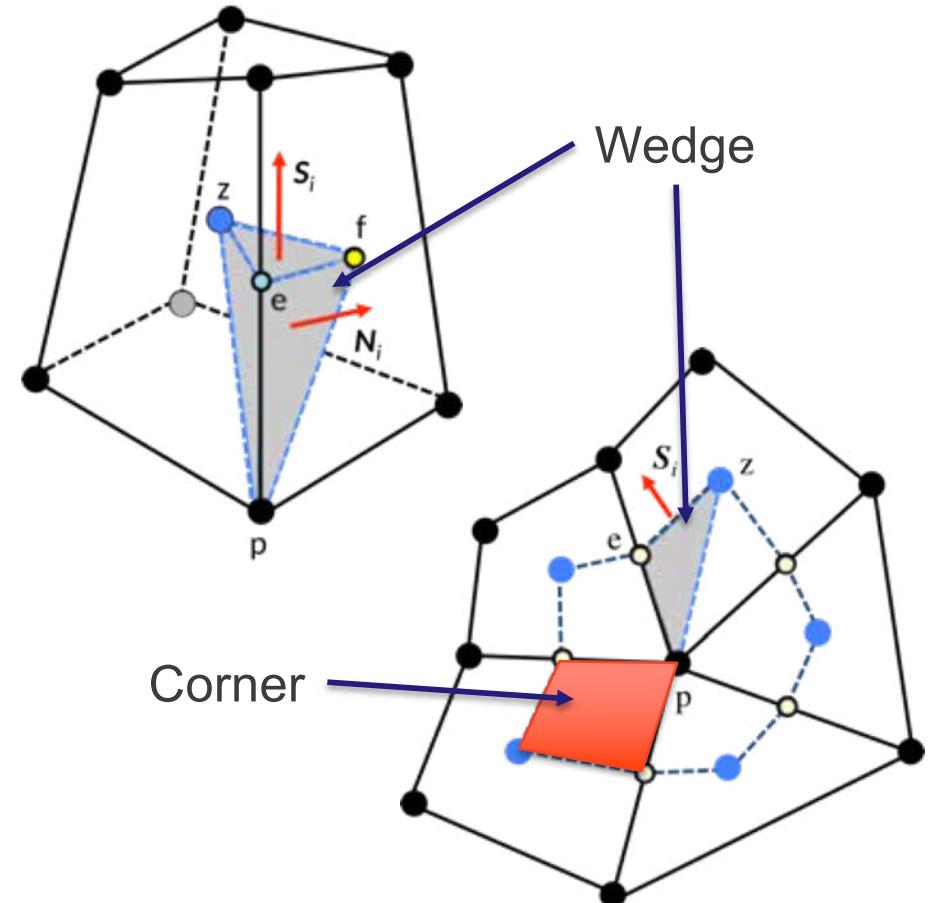
3D: `burton_mesh_t<3> mesh;`

Application code doesn't change (code works in 2D and 3D):

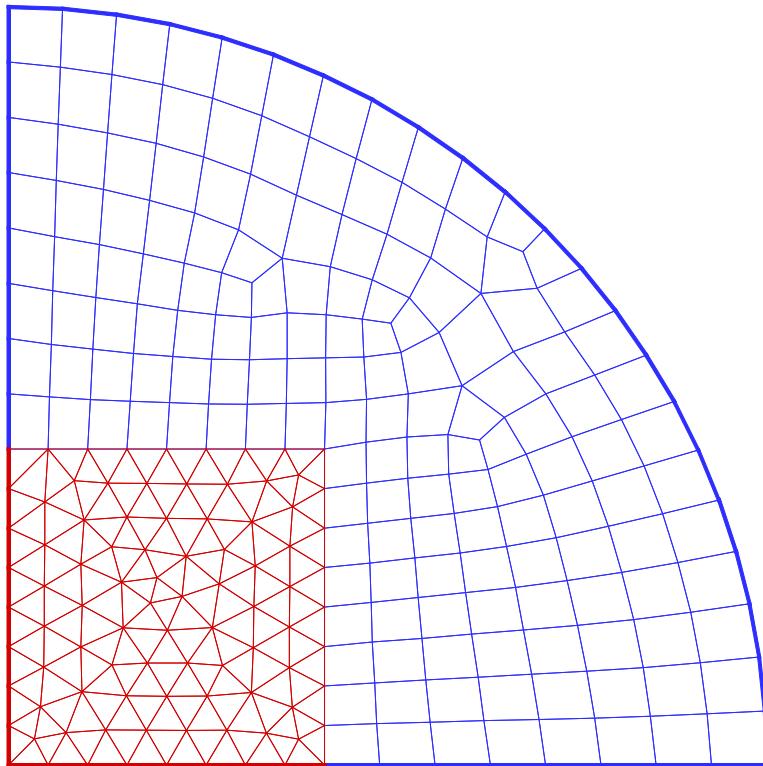
```
for ( auto f : mesh.faces() )
    auto n = f->normal();
    // do some work
```

Mesh has wedges and corner data structures in addition to vertex, edge, face, and cell primitives:

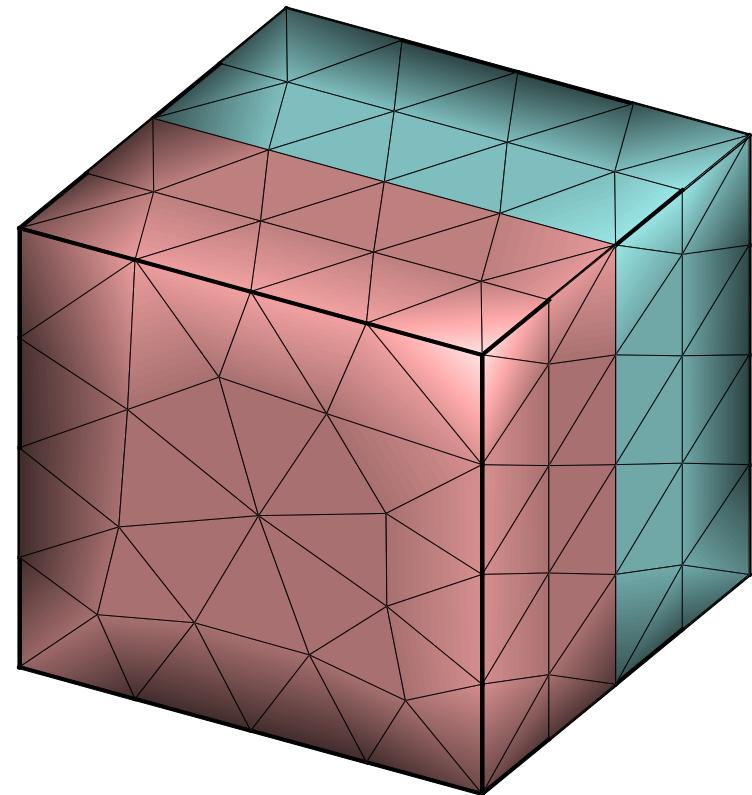
```
for ( auto cn : mesh.corners() )
    for ( auto wg : mesh.wedges(cn) )
        auto n = wg->facet_normal();
        // do some other work
```



Supports mixed element types and multi-block meshes

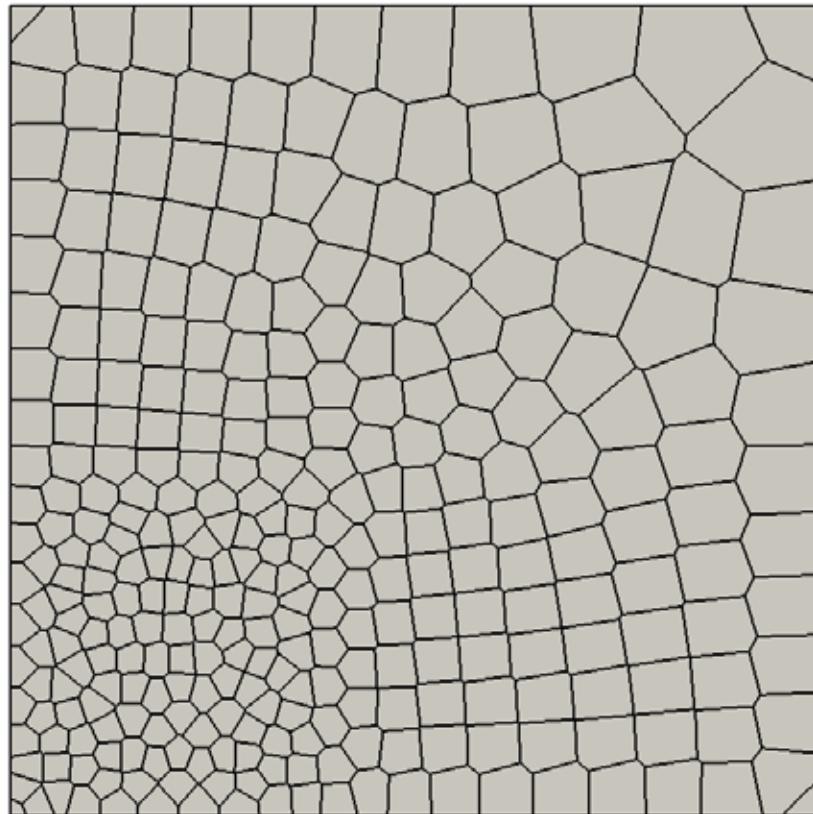


Mixed triangles and quads

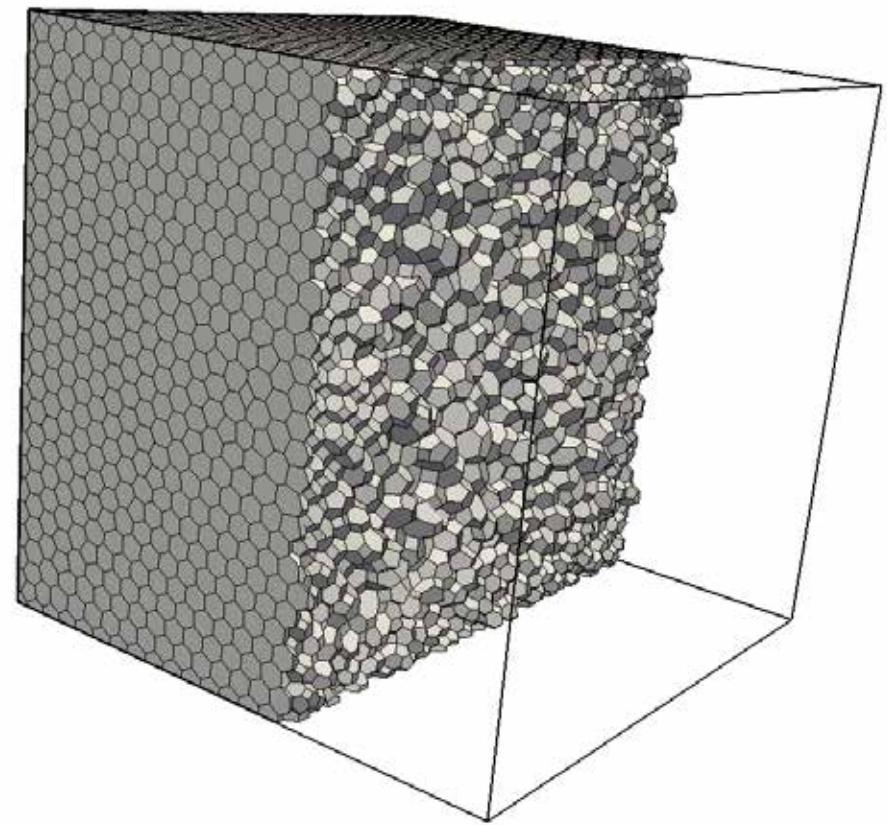


Multiblock unstructured meshes

Supports arbitrary polygons and polyhedra

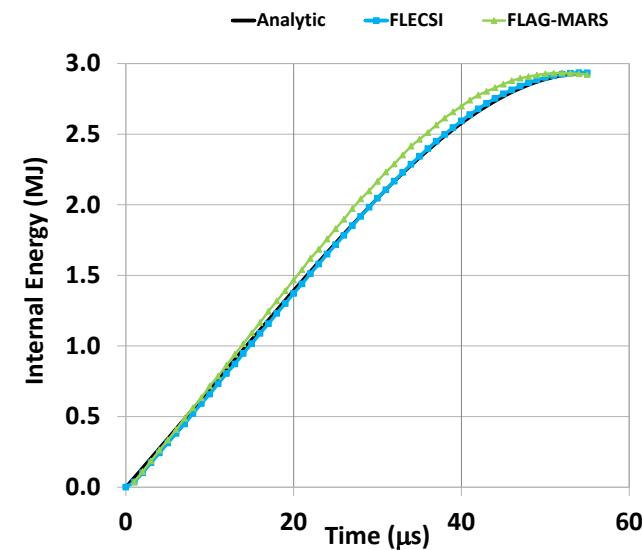


Arbitrary polygons



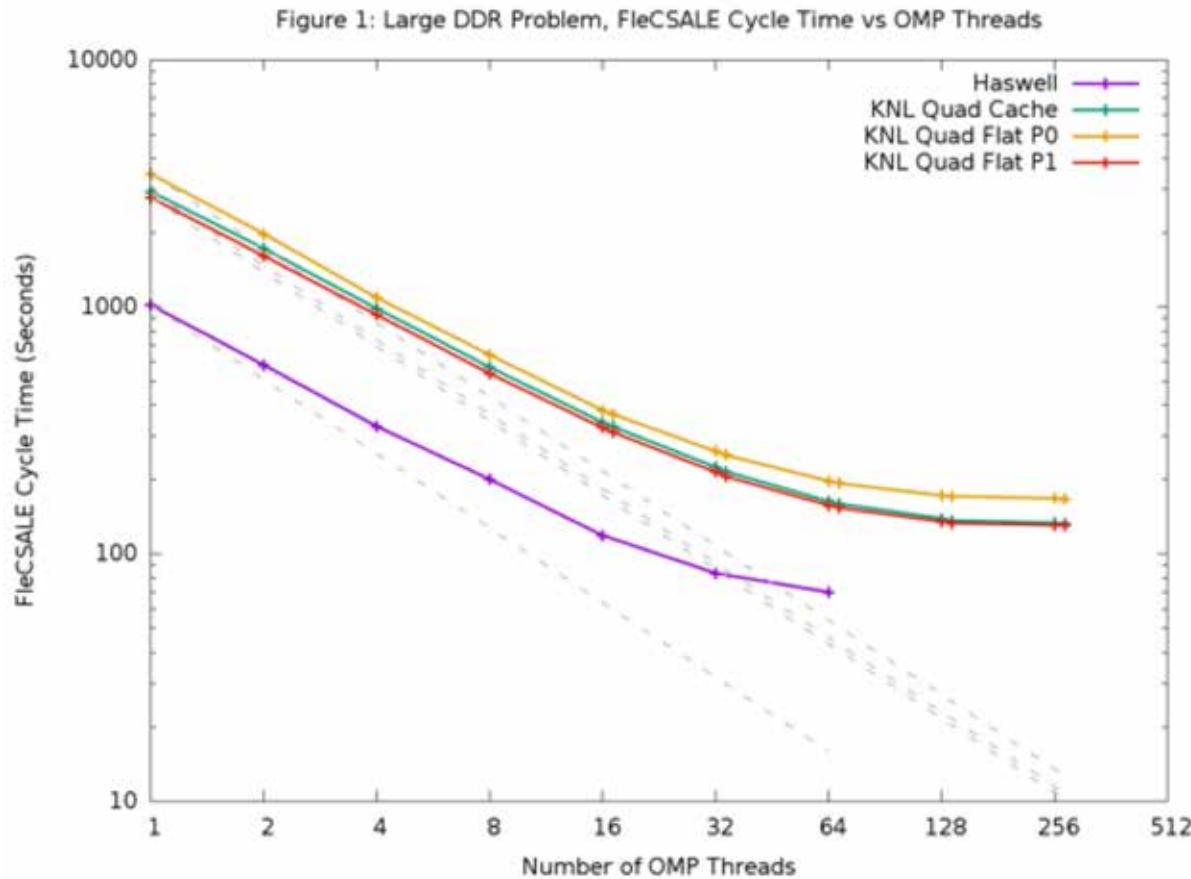
Arbitrary polyhedra

Predictions of the Verney steel shell problem using Lagrangian FEM



Calculated internal energy.

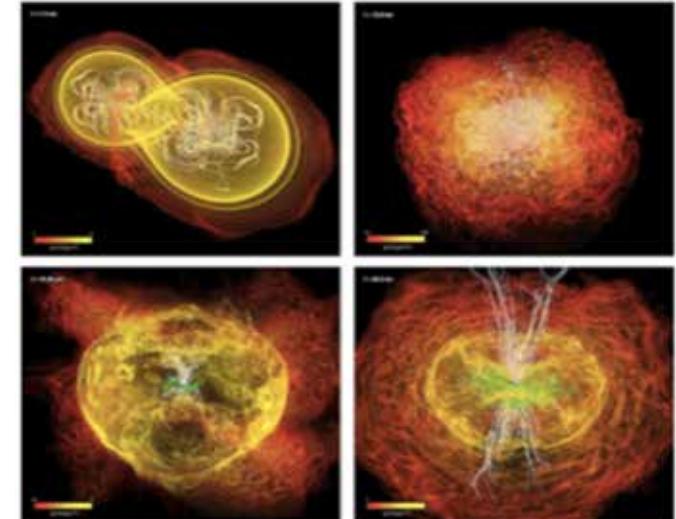
FleCSALE: Strong Scaling on Haswell & KNL



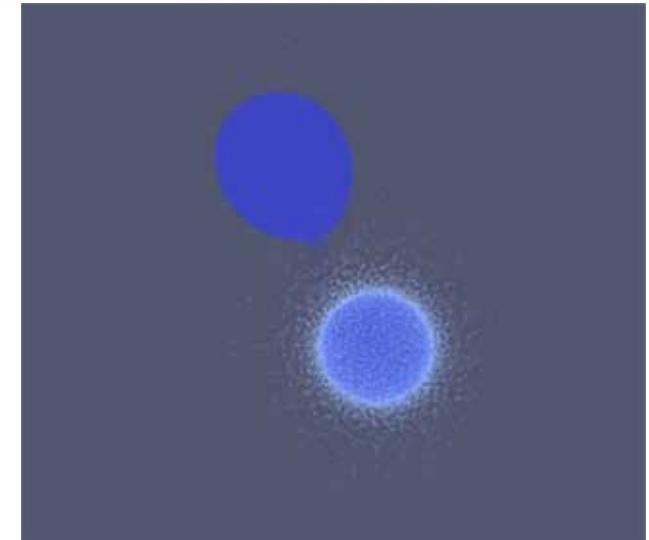
FleCSPH

What is FleCSPH?

- **FleCSPH is a smoothed-particle hydrodynamics (SPH) solver built on the FleCSI programming system.**
 - Utilizes FleCSI hashed-Octree topology
 - Lagrangian conservation equations for mass, momentum, and total energy
 - Ideal fluids with Newtonian gravity
 - Distributed-memory parallel using direct MPI
- Developed by summer GRA students Julien Loiseau and Hyun Lim with science leads Oleg Korobkin and Wes Even
 - Project started in April 2017
- Initial focus on astrophysics simulations of neutron star mergers
 - Potential source of gravitational waves, macronovae, and nucleosynthesis



Rezzola et al. (2011)



Neutron Star Mergers

- **Kilonovae (macronovae) transients emit isotropically**
- **Better detection for gravity wave detectors (poor directional sensitivity)**

Simulations seek to increase understanding of kilonovae properties

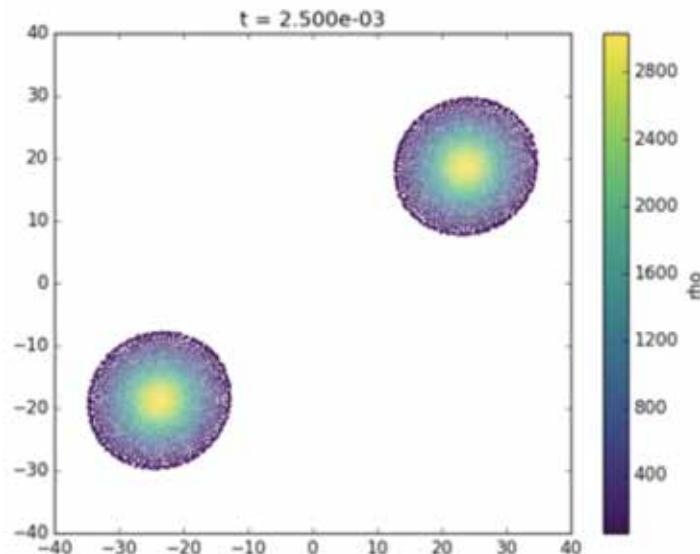
- **Ejecta Mass**
 - Amount of mass ejected through different channels (dynamical, neutrino-driven winds, viscous winds)
- **Ejecta Morphology**
 - Structural form of ejecta
- **Ejecta Composition**
 - Relative densities of heavy *r*-process elements



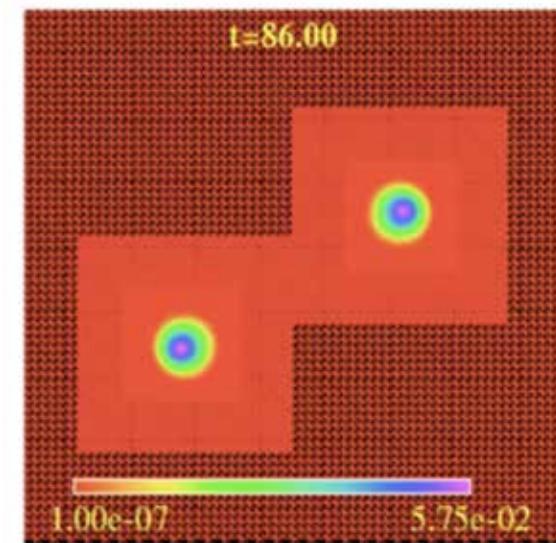
Powered by the decay of *r*-process elements (synthesized in the merger ejecta), kilonovae can peak in the infrared or visual bands on the timescale of a few days

Smoothed-Particle Hydrodynamics (SPH)

**Exact conservation of mass, linear & angular momentum,
and total energy**



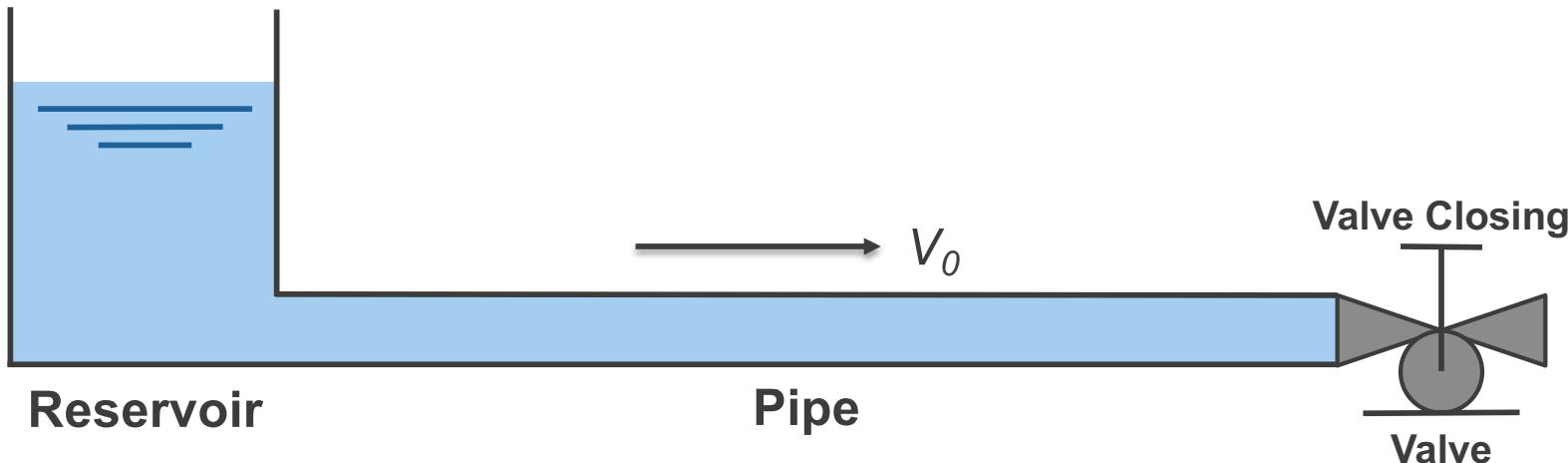
**Particle method naturally handles
vacuum and deformations**



**Mesh-based methods require
artificial atmosphere**

Water Hammer Problem

A water hammer is a pressure surge or wave caused when a fluid (usually a liquid but sometimes also a gas) in motion is forced to stop or change direction suddenly.



Simulations seek to understand pressure and velocity extremes in relevant parts of the model

Future Work

- **FleCSI features**
 - Nested data handles
 - Complete storage type parallel implementations (sparse & tuple)
 - Improved execution & data handling for tree topologies
 - Improvements to static analysis and compile tool: *flecsit*
- **Node-level, fine-grained data-parallel interface**
 - Investigation of embedded DSL using Clang/LLVM toolchain as part of the FY18 Co-Design L2 Milestone

```
foreach(auto c: mesh.cells(owned)) {  
    p(c) = ...  
} // foreach
```

- **Integration of new Legion features**
 - Control Replication
 - Dependent Partitioning
- **Backend runtime support**
 - Charm++ & HPX contracts are in negotiation

Thanks for listening!